Extracted from:

Test iOS Apps with UI Automation

Bug Hunting Made Easy

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Test iOS Apps with UI Automation

Bug Hunting Made Easy

Jonathan Penn

Edited by Brian P. Hogan

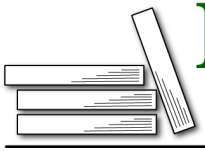
Test iOS Apps with UI Automation

Bug Hunting Made Easy

Jonathan Penn

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-52-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2013

5.2 Identifying Elements with Accessibility APIs

Our test works fine for the moment, but we have a fragile situation on our hands. As we see in Figure 26, *Finding the map pins in the window*, on page ?, the pin annotation views are represented as just plain UIAElement instances. Although we're able to use the special filtering methods such as buttons() and cells() to fetch only elements of a certain type, we don't have that convenience here. We have to use the generic elements() method.

This could be a problem if we were testing for a pin with the name Legal, for instance, because the static text element for the Map Kit legal disclaimer has that name and is a sibling in the element array, along with all our pins. This could give us a false positive in a test. We need a better way to uniquely identify our pins.

To address this, we're going to experiment with the accessibility APIs to understand how UI Automation sees UIAElement instances. We'll be jumping back and forth between Objective-C and JavaScript a bit, but it will pay off because we'll have a much more reliable way to distinguish our pins from the other elements on the screen.

The accessibility APIs are the basis for technologies like VoiceOver that give visually impaired users a better experience. You can specify traits to identify elements as buttons, search fields, adjustable controls, or anything else that a user would need to manipulate on the screen.

We will use the accessibility APIs to define identifiers for our test's use that are separate from what the user can see and hear. By conforming to the UIAccessibilityIdentification *informal protocol*, any UIView subclass can change the value of the name() method on its UIAElement representation. This protocol is informal because we don't have to declare it in the Objective-C class interface. We merely have to define a method with the name accessibilityIdentifier in a subclass.

Since Map Kit gives us full control over the annotation views that appear over the map, we can use a custom subclass of MKPinAnnotationView. Let's switch back to Xcode and change our map results view controller so it returns our subclass:

05-MapsGestures/step03/NearbyMe/NBMMapResultsViewController.m

```
- (MKAnnotationView *)mapView:(MKMapView *)mapView
            viewForAnnotation:(id<MKAnnotation>)annotation
{
    if ([annotation isKindOfClass:[MKUserLocation class]]) {
        return nil;
```

```
    } else {
        MKPinAnnotationView *view = [[NBMAccessibleAnnotationView alloc]
                                     initWithAnnotation:annotation
                                     reuseIdentifier:nil];

        view.canShowCallout = YES;
        view.animatesDrop = YES;
        return view;
    }
}
```

As the delegate of the map view, this view controller will be asked for an annotation view for every annotation on the map. We first check to see if the given annotation represents the user's current location. If it does, we're just returning nil so the Map Kit framework can handle it the way it normally does, with the blue pulsating dot. If this isn't a user-location annotation, then we create an instance of our custom subclass, NBMAccessibleAnnotationView, and return it instead.

In our custom annotation view we return a new string for the accessibilityIdentifier:

05-MapsGestures/step03/NearbyMe/NBMAccessibleAnnotationView.m

```
@implementation NBMAccessibleAnnotationView

- (NSString *)accessibilityIdentifier
{
    NBMPointOfInterest *poi = self.annotation;
    return [NSString stringWithFormat:@"POI: %@", poi.title];
}

@end
```

We use the annotation that belongs to this annotation view and return a string of the annotation title prefixed with POI:. Remember that the accessibility identifier is not visible or audible to the user. We can put whatever we want in here to distinguish our elements.

We need to rebuild our application and load it in Instruments so our automation scripts can see this change. Choose Profile from the Product menu in Xcode or press ⌘-I, and the app will build and open in the Instruments trace document.

In the UI Automation script editor, let's write a quick script in our sandbox file that takes us to the map screen and logs the element tree so we can see what changed:

05-MapsGestures/step03/automation/sandbox.js

```
#import "env.js";

var target = UIATarget.localTarget();
var window = target.frontMostApp().mainWindow();
SearchTermScreen.tapTerm("coffee");
target.delay(5);    // Adjust to give network time to fetch
window.logElementTree();
```

We're tapping the search term and pausing for a moment so the network request can complete before logging the element tree. When we run our automation script by pressing ⌘-R twice to stop and restart the trace, we'll see output similar to what the following figure shows. This prefix gives us a reasonable way to uniquely distinguish points of interest from other elements.

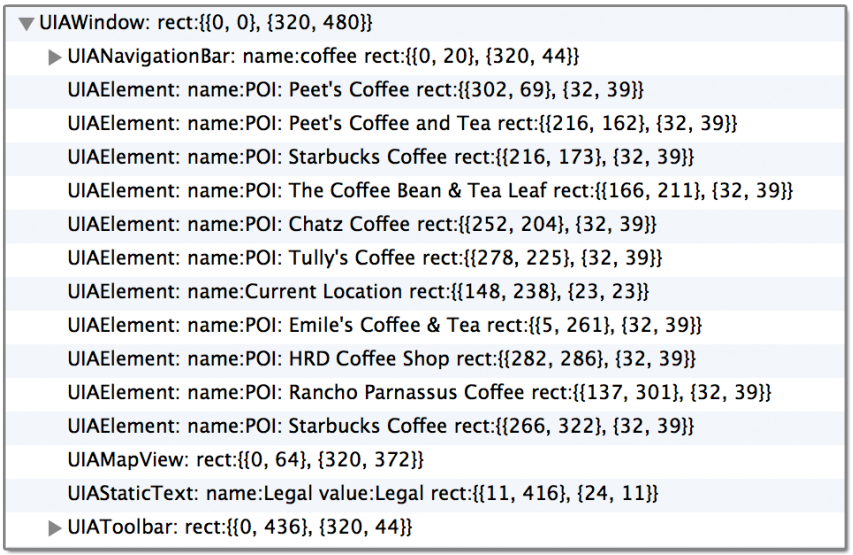


Figure 27—Prefixing POI: to pin identifiers

We need to change how our ResultsMapScreen object works, because it doesn't know about the POI: prefix yet. Let's do this by building a method specifically for looking up pins by name on the map:

05-MapsGestures/step03/automation/lib/screens/ResultsMapScreen.js

```
pinNamed: function(name) {
    log("Looking up", name, "on the map");
    var elements = this.window().elements();
    return elements["POI: " + name];
}
```

We're using simple string concatenation to add the prefix to the name passed in to our assertion. Since the ResultsMapScreen object encapsulates all the logic for looking up a point of interest by name, we can change the prefix of pin identifiers again if we need to. As long as we've used our screen objects in all the tests we write, we only have to make the update for the new pin identifiers in this one file.

Now we can update our assertion to use the new method:

05-MapsGestures/step03/automation/lib/screens/ResultsMapScreen.js

```
assertPinNamed: function(name) {
    assert(this.pinNamed(name).isValid(), "Not found");
},
```

Run the whole test suite again to make sure nothing is broken; everything passes!

The accessibility APIs are a powerful ally when working with automation scripts. Changing the accessibilityIdentifier on view objects is a great way to help solve ambiguity. There are many more ways to use these APIs to your advantage, such as by defining containers of elements for nonview objects and hiding views from the element tree entirely. Adapting these representations to your application improves the user experience and makes it easier to access elements in your tests. Check Apple's documentation for more information.[1]

Now that we've studied the map and how elements are represented, we're ready to start writing our final acceptance test for this chapter and learning a bit about simulated gestures along the way.

1. http://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/iPhoneAccessibility/Accessibility_on_iPhone/Accessibility_on_iPhone.html